[14] Steven P. Reiss and John T. Stasko. The Brown Workstation Environment: A user interface design toolkit. In Gilbert Cockton, editor, *Engineering for Human-Computer Interaction*, pages 215–228, Amsterdam, 1990. North Holland. Proceedings of the IFIP WG 2.7 Working Conference on Engineering for Human-Computer Interaction, Napa Valley, CA, Aug. 1989.

[15] Robert W. Schiefler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[16] Donald E. Knuth. *TEXand METAFONT. New Directions in Typesetting.* Digital Press, Bedford, MA, 1979.

References

[1] Steven Feiner, David Salesin, and Thomas Banchoff. Dial: A diagrammatic animation language. *IEEE Computer Graphics and Applications*, 2(7):43—54, September 1982.

[2] Craig W. Reynolds. Computer animation with scripts and actors. *Computer Graphics: SIGGRAPH '82*, 16(3):289–296, July 1982.

[3] E. Fiume, D. Tsichritzis, and L. Dami. A temporal scripting language for object-oriented animation. In D. Tsichritzis, editor, *Objects and Things*, pages 129–141. Universite De Genève, 1987.

[4] Kavi Arya. A functional approach to animation. *Computer Graphics Forum*, 5:297–312, 1986.

[5] S. Hayward. *Computers for Animation*. Focal Press, Boston, MA, 1984.

[6] Ronald M. Baecker. Picture-driven animation. In *Spring Joint Computer Conference*, volume 34, pages 273–288. AFIPS Press, 1969.

[7] William R. Mallgren. Formal specification of graphic data types. *ACM Transactions on Programming Languages and Systems*, 4(4):687–710, October 1982.

[8] E. Fiume. An attempt at formal specifications for a non-trivial object. In D. Tsichritzis, editor, *Objects and Things*, pages 149–164. Universite De Genève, 1987.

[9] John T. Stasko. *TANGO: A Framework and System for Algorithm Animation*. PhD thesis, Brown University, Providence, RI, May 1989. Available as Technical Report No. CS-89-30.

[10] Marc H. Brown. Exploring algorithms using Balsa-II. *Computer*, 21(5):14–36, May 1988.

[11] Robert A. Duisberg. Animated graphical interfaces using temporal constraints. In *Proceedings of the ACM SIGCHI '86 Conference on Human Factors in Computing Systems*, pages 131–136, Boston, MA, April 1986.

[12] Esa Helttula, Aulikki Hyrskykari, and Kari-Jouko Räihä. Graphical specification of algorithm animations with Aladdin. In *Proceedings of the 22nd Hawaii International Conference on System Sciences*, pages 892–901, Kailua-Kona, HI, January 1989.

[13] John T. Stasko. TANGO: A framework and system for algorithm animation. *Computer*, 23(9):27–39, September 1990.

animated views of program executions. We illustrated sample TANGO code from an animation of the Towers of Hanoi problem.

The development of an animation paradigm/system is never really finished because there are always more features to be added and improvements to be made. Much of this is suggested by using the system. Below are capabilities that we plan to add to the paradigm in the future:

- A *bitmap* image type that would allow designers to hand-craft a graphical object and a *shuffle* transition for displaying series of these bitmaps.

- Improved definitional facilities for composite images that would allow the individual subimages to be named and referenced for use in transitions. The subimages should also be parameterized so that they can be modified at run-time.

- More sophisticated methods for designing paths—in particular, methods similar to those used in the Metafont system[16] for creating arcs using intermediate points and enter and exit angles.

- Strengthened support for the layout and organization of complex, structured imagery, possibly through the inclusion of constraint specifications to help maintain graphical relationships within dynamic structures.

In terms of language support, implementing the paradigm in a more rigorous object-oriented environment such as C++ offers appeal. In TANGO we chose not to utilize the object-based appearance of the paradigm for local, pragmatic reasons—very few of our potential animation designers had experience with C++. Another future direction we plan to pursue is a definition of a formal, stand-alone algorithm animation language with its own specialized syntax tailored to the requirements of algorithm animation. Our initial experience of using C as a backbone environment has provided the foundation for creating a comprehensive animation language definition.

Acknowledgments

Steve Reiss, Jeff Vitter, Paris Kanellakis, Andy van Dam, Marc Brown, Rob Rubin, and the paper's referees all contributed to this work with insightful comments and suggestions.

```
   for (i=0; i<disks; ++i)      /* set up the association for disks residing at locations */
      { peg_pos = AssocRetrieve("POINTS",ID,i,0);
        disk = AssocRetrieve("DISK",ID,i);
        outline = AssocRetrieve("OUTLINE",ID,i);
        AssocStore("DISK_AT",peg_pos, disk);   /* final parameter is value to be stored */
        AssocStore("OUTLINE_AT",disk, outline);
      }

   path = PathNull(3);
   trans = TransCreate(Delay,NULL,path);        /* make everything appear */
   TransPerform(trans);
}


/* given there are fromdisks on peg number frompeg, move the top disk onto */
/* peg number topeg which currently has todisk number of disks on it.  */

void
Move(frompeg,fromdisk,topeg,todisk)
   int frompeg,fromdisk,topeg,todisk;
{
   int        top_spot,open_spot;
   Location   atpt,frompt,topt,topeg_top,frompeg_top;
   Image      disk,rect;
   Path       path[3],movepath;
   Trans      move[2],compose;

   top_spot = num_disks - fromdisk;                          /* find top disk on peg frompeg (reverse) */
   atpt = AssocRetrieve("POINTS",ID,top_spot,frompeg);       /* acquire that location */
   disk = AssocRetrieve("DISK_AT",atpt);                     /* get the disk currently there */
   rect = AssocRetrieve("OUTLINE_AT",disk);                  /* get the outline too */

   frompt = ImageLocate(rect,South);                         /* acquire locations for disk movement */
   frompeg_top = AssocRetrieve("PEGTOP",ID,frompeg);
   topeg_top = AssocRetrieve("PEGTOP",ID,topeg);
   open_spot = num_disks - todisk - 1;                       /* find loc on topeg to put this disk */
   topt = AssocRetrieve("POINTS",ID,open_spot,topeg);

   path[0] = PathDistance(frompt,frompeg_top,OFF_DISTANCE);  /* make path to move up, across, and down */
   path[1] = PathDistance(frompeg_top,topeg_top,OFF_DISTANCE); /* OFF_DISTANCE is a macro */
   path[2] = PathDistance(topeg_top,topt,OFF_DISTANCE);
   movepath = PathConcatenate(3,path[0],path[1],path[2]);

   move[0] = TransCreate(Move,disk,movepath);                /* move disk and its outline */
   move[1] = TransCreate(Move,rect,movepath);
   compose = TransCompose(2,move[0],move[1]);
   TransPerform(compose);

   AssocStore("DISK_AT",topt, disk);                         /* reset the disk to its new location */
}
```

Figure 7: Animation scenes *Init* (continued) and *Move* for the Towers of Hanoi animation.

```
#define  ID            0          /* ID used in associations */
#define  DISK_HT       0.1        /* y value--disk heights   */
#define  OFF_DISTANCE  0.03       /* used in creating movement paths */

static int num_disks;

/* draw the 3 pegs, create a two-dimensional array of positions for the  */
/* disks to go onto the pegs, and draw the initial set of disks on peg 0 */

void
Init(disks)
   int disks;  /* number of disks to start on first peg */
{
   int          i;
   double       xvals[10], stack_top, peg_ht;
   Location     peg_pos;
   Image        disk,outline;
   Path         path;
   Trans        trans;

   AssocCreate("PEG",2);             /* peg images themselves */
   AssocCreate("PEGTOP",2);          /* 3 locations above the tops of the pegs */
   AssocCreate("POINTS",3);          /* 2-d array of locations */
   AssocCreate("DISK",2);            /* actual disk images */
   AssocCreate("OUTLINE",2);         /* actual outline images */
   AssocCreate("DISK_AT",1);         /* get a disk given a location */
   AssocCreate("OUTLINE_AT",1);      /* get a disk's outline, given the disk */

   num_disks = disks;
   peg_ht = (disks+1) * DISK_HT;     /* scale the 3 pegs that hold disks */

                /* Draw the pegs */
   TwistCreateImageArray("PEG",ID,3,LINE,1,0,0.2,0.9,NULL,0.0,NULL,peg_ht,0.3,1,Black,0.0,0.0,1.0);
                /* Params are (AssocName,AssocID,#,type,horiz?,justif,xpos,ypos,xvals,xscale,
                               yvals,yscale,spacing,vis,color,fill,width,style)    */

                /* Designate tops of pegs for use in disk movement */
   TwistCreateLocArray("PEGTOP",ID,3,1,0.2,0.9-peg_ht,0.3);
                /* Params are (AssocName,AssocID,#,horiz?,xpos,ypos,spacing)  */

   stack_top = 0.9 - (disks * DISK_HT);    /* calc x value of top of disk stack */

                /* make 2d Array where each column is a peg and each row position can hold a disk */
   TwistCreate2dLocArray("POINTS",ID,disks,3,0.2,stack_top+DISK_HT,0.3,DISK_HT);
                /* params are (AssocName,AssocID,rows,columns,xpos,ypos,xspacing,yspacing)  */

   for (i=0; i<disks; ++i)      /* create array of scaled disk widths */
      xvals[i] = (double)(i+1) / (double)disks;

       /* put the colored disks (and black outlines) on the first peg */
   TwistCreateImageArray("DISK",ID,disks,RECT,0,1,0.2,stack_top,xvals,0.3,NULL,DISK_HT,0.0,1,Yellow,1.0,0.0,0.0);
   TwistCreateImageArray("OUTLINE",ID,disks,RECT,0,1,0.2,stack_top,xvals,0.3,NULL,DISK_HT,0.0,1,Black,0.0,0.0,0.0);

   /* continued */
```

Figure 6: Animation scene *Init* for the Towers of Hanoi animation (beginning).

Figure 5: Animation view for the Towers of Hanoi problem.

also sets up associations so that the created images can be referenced in subsequent scenes.

7 Conclusion

We have introduced the path-transition paradigm for designing two-dimensional, color animations. The paradigm is based upon four abstract data types: location, image, path, and transition. Each data type includes a set of operations that facilitate the creation of sophisticated animation actions. We defined the semantics and specifications of the operations through an informal algebraic approach. Operations modify the animation state with the special *Perform* transition operation adding image configurations to the list of animation frames.

We also described a prototype implementation of the paradigm, the TANGO algorithm animation system, which illustrates that the paradigm is both practical and powerful through its

disk on the bottom, the smallest disk on top, and intermediate size disks in between. The sizes of the disks increase from the top of the stack to the bottom. The problem is to move all the disks to one of the empty pegs subject to the following two constraints:

- Only one disk can be moved at a time.
- A disk can only be placed on top of a disk larger than itself.

Using TANGO we designed an animation of the problem that begins showing the three pegs with all the disks on the first peg. The animation proceeds by having each transfer of a disk simulated by moving the disk's image up off its peg, across to its target peg, and then down to its new position on that peg. Image movements are gradual; the use of many "in-between" frames creates the illusion of smooth, continuous movement. Figure 5 illustrates a sample frame from the animation view.

Our implementation of the animation consists of only two animation scenes, shown in Figures 6 and 7. The first scene, *Init*, is called once at the start of the animation, and it displays the n disks on the first peg. The second scene, *Move*, shows a disk moving from one peg to another. It receives parameters specifying the source and destination peg numbers, as well as the number of disks on each.

The basic idea of the animation is that we create a 2-d array of locations for all possible disk positions on the three pegs. Rows in the array correspond to the positions of disks stacked on a peg, and the columns correspond to the three pegs themselves. Given row and column indices, we can acquire the corresponding location instance. We also set up a dynamic association, IMAGE_AT, that is used for placing and then acquiring a disk image with respect to a particular location. When the *Move* scene receives the *from* and *to* pegs plus the numbers of disks on each, we access the two particular locations of the *from* and *to*, and we acquire the disk currently on the *from* location. We then set up a path that can be used in a *move* transition to guide the disk to the *to* location.

Notice that the *Init* animation scene includes references to "Twist" routines. These are a set of macro routines we have built to aid animation designers, and they simply consist of structured data type operation (TANGO) calls. For example, the routine TwistCreateImageArray creates a row or column of images; parameters to the routine specify attributes such as the number of images, image type, geometric placement and spacing, color, scale factors for sizing, and so on. The routine

6 Animation Example

In this section we will present an example animation in order to illustrate how the path-transition paradigm facilitates animation design. We begin by describing an additional facility that we added to the paradigm particularly for animating programs. This facility derives from the notion of an algorithm animation as a representation of a process, a mapping from the programming world into a visual world. In algorithm animation the animated view is a simulation of the abstract operations of a computer program.

In the path-transition paradigm the connection facility from program objects to animation objects is provided through *associations*. An association allows a designer to connect a data object such as an image, location, or data value of importance to a set of parameters received from a driving program or process. Associations strictly between data type instances can also be created. Associations are identified by unique names, and they can utilize zero or more parameters.

More formally, an association is a method to store a data object such as an animation data type object or a data value. A typical implementation uses hashing. The hash key for the data object is the name of the association and a list of parameters. The number of parameters to each association is fixed. Typically, the parameters are the data values being mapped from a program. For example, to associate the elements of an integer array a of length 10 with ten rectangle images, we could create an association with name "IMAGE", first parameter the initial memory address of a, and second parameter the particular array index position.

Three functions perform all of the association duties within the paradigm. The *Create* function defines an association and the number of parameters it will use. An association must be defined with this function before it can be utilized. The *Store* function associates a data object with a key—the association's name and a list of parameters. Attempts to store a data object in an association with the incorrect number of parameters will be considered an error; no data will be stored. The *Retrieve* function returns the data object associated with a given key. If no data object has been stored with the given key, a null value is returned.

The example animation we will present is a simulation of the Towers of Hanoi challenge, a classic problem used to illustrate recursion in teaching computer science. The problem utilizes three pegs and a number of disks of different sizes. The disks all begin in a stack on one peg with the largest

5 Algorithm Animation Application

The initial testing ground for our animation paradigm is an *algorithm animation* system. Algorithm animation is the process of abstracting the data, operations, and semantics of computer programs, and then creating dynamic graphical views of those abstractions. Algorithm animation is more comprehensive than simple data structure rendering which typically exhibits one-to-one mappings between program data and graphical objects. Algorithm animation involves program views that conceptually portray how a program works and, frequently, have no direct correspondence to the program's data or execution units. To date, algorithm animations primarily have been used as instructional aids, but we are currently exploring their utility as graphical debugging aids also.

Some of the better known systems for algorithm animation are Balsa[10], Animus[11], and Aladdin[12]. Our system is called TANGO (Transition-based ANimation GeneratiOn) and it supports two-dimensional, color, real-time animations in a window-based workstation environment[9, 13]. In TANGO we implemented the four abstract data types of the path-transition paradigm as user-defined types in the C programming language. We also implemented their operations as a package of calls available to animation designers. Our implementation includes line, rectangle, circle, ellipse, polyline, polygon, spline, and text image types, as well as move, resize, color, fill, visibility, raise, lower, and delay transitions. To implement a TANGO algorithm animation, a designer supplements the program being animated with events that we call algorithm operations. As program execution occurs, the algorithm operations are sent out as interprocess messages through a central message server and on to TANGO animation windows. The messages are decoded by TANGO which then activates the appropriate animation routines, the designer-written animation scenes that use the data types and operations of our animation paradigm.

TANGO currently runs on a network of Sun and DEC workstations and is supported by the Brown Workstation Environment[14], an application interface toolkit built on top of the X11 window system[15]. TANGO has been used to animate programs from a wide variety of domains such as sorting, searching, hashing, and graph and tree manipulations, as well as animated simulations such as a producer-consumer ring buffer and the post office queuing problem. The system is also being used as an instructional aid in computer science courses.

The *apply* operation receives a transition unit which we denote as (*transtype*, *image*, x, y). It performs two main functions.

First, it affects the animation state by modifying the image's parameters (the image is a member of the I field of the animation state) as dictated by the transition type and the image type. For instance, given that *im* is a circle,

apply((*move*, *im*, x', y'), C)
$\Rightarrow$ *im*:[circle, (loc.x+x', loc.y+y'), vis, radius, color, fill] $\in I$,

apply((*resize*, *im*, x', y'), C)
$\Rightarrow$ *im*:[circle, loc, vis, radius+x', color, fill] $\in I$,

and so forth for each of the different transition types.

Second, when given a transition unit of type *raise*, the *apply* operation removes the image from its current location in the configuration and brings it to the front of the list. The *lower* transition works in the expected opposite manner.

apply((*raise*, *im*, x', y'), C) $\Rightarrow$ $C - im$; cons(im, C).

Briefly reviewing, all operations and manipulations within the paradigm occur in order to produce animation. We introduced the concept of a configuration, an ordered list of the active images, in order to characterize the relative ordering of image objects. It allows us to formally define an animation as a list of configurations.

The animation process begins with an empty list of configurations. As animation scenes are executed, individual data type operations modify the animation state and the current configuration as described in this section. The special *Perform* transition operation adds configurations as elements in the animation list as follows:

$AnimList$ = append($AnimList$, $Perform(tr)$).

$AnimList$ begins as a null list, and it grows each time *Perform* is called. After the final animation scene is processed, we are left with the list of configurations characterizing the particular animation that has occurred.

```
mcopy(mom) → if (mom) then
                 cons( (car(mom).tu, car(mom).im, car(mom).x, car(mom).y),
                       mcopy(cdr(mom)) )
             else
                 ∅.
```

Definitions of the transition operations follow:

```
TransConcatenate(tr1, tr2) → append(tcopy(tr1), tcopy(tr2)).

TransIterate(tr, i) → if (i > 0) then
                          append( tcopy(tr), TransIterate(tr, i − 1) )
                      else
                          ∅.

TransCompose(tr1, tr2) → if (tr1 and tr2) then
                             cons( append(mcopy(car(tr1)), mcopy(car(tr2))),
                                   TransCompose(cdr(tr1), cdr(tr2)) )
                         else if (tr1) then
                             tcopy(tr1)
                         else if (tr2) then
                             tcopy(tr2)
                         else
                             ∅.
```

Once a transition has been created via any of the above operations, it is carried out with the special *Perform* operation. As discussed earlier, *Perform* may modify the animation state and rearrange the configuration. Additionally, it is the only operation that adds configurations to the algorithm animation list, actually generating animation frames. The functionality of *Perform* is detailed below. The symbol C refers to the current configuration.

```
TransPerform(tr) → if (tr) then
                       moment = car(tr)
                       while (moment) do
                           { trans_unit = car(moment)
                             apply(trans_unit, C)          /* see below */
                             moment = cdr(moment)
                           }
                       return( cons( C, TransPerform(cdr(tr)) ) )
                   else
                       return ∅.
```

Data type Transition

Operations

∘TransCreate:	transtype × image × path	→ transition;
TransConcatenate:	transition × transition	→ transition;
TransIterate:	transition × integer	→ transition;
TransCompose:	transition × transition	→ transition;
TransPerform:	transition	→ ⟨⟨ image ⟩⟩.

Preliminaries

Below we define how a transition is generated by the *Create* operation. The operation produces a list of moments, which in this case, are each only one transition unit. Assume that im = ImageCreate($itype$, bo, lo, $\{params\}$), and pa = PathCreate($\langle lo \rangle$).

TransCreate($ttype$, im, pa)
→ if (pa) then
cons(cons(($ttype$, im, LocX(car(pa)), LocY(car(pa))), ∅),
TransCreate($ttype$, im, cdr(pa)))
else
∅.

Definitions

Utilizing the model of a transition as a list of moments, we now define how the three transition operators combine transitions to make new transitions. This list format of transitions is especially convenient for the following definitions. The transition arguments to the operators are simply lists of moments, and they must have been created by the *Create* operation or one of the three operations being defined here. Of course, some original transition must be defined using *Create* to begin the process. For brevity in presentation purposes, assume that tr = TransCreate($ttype$, im, pa) and that in the transition operations below, the functions tcopy (transition copy) and mcopy (moment copy) are defined as follows:

tcopy(tr) → if (tr) then
cons(mcopy(car(tr)), tcopy(cdr(tr)))
else
∅.

Given that $mom = \langle tu \rangle$,

Data type Path

Operations

∘PathCreate:	⟨location⟩	→ path;
PathLength:	path	→ integer;
PathDx:	path	→ real;
PathDy:	path	→ real;
PathNull:	integer	→ path;
PathMakeType:	pathtype	→ path;
PathCopy:	path	→ path;
PathRotate:	path × integer	→ path;
PathAddHead:	path × integer	→ path;
PathAddTail:	path × integer	→ path;
PathDeleteHead:	path × integer	→ path;
PathDeleteTail:	path × integer	→ path;
PathScale:	path × real × real	→ path;
PathExtend:	path × real × real	→ path;
PathInterpolate:	path × real	→ path;
PathColor:	color	→ path;
PathDistance:	location × location × real	→ path;
PathExample:	location × location × path	→ path;
PathConcatenate:	path × path	→ path;
PathIterate:	path × integer	→ path;
PathCompose:	path × path	→ path.

Sample definitions

For reasons of brevity, we will substitute *pa* for the basic generator object PathCreate(⟨*lo*⟩) as the argument in the following operations, and we will utilize the notion of a path as an ordered list of offsets (location objects).

PathLength(*pa*) → if (*pa*) then
1 + PathLength(cdr(*pa*))
else
0.

PathScale(*pa*, *x*, *y*)
→ if (*pa*) then
cons(LocCreate(*x*· LocX(car(*pa*)), *y*· LocY(car(*pa*))),
PathScale(cdr(*pa*), *x*, *y*))
else
∅.

Data type Location

Operations

∘LocCreate:	real × real	→ location;
LocModify:	location × real × real	→ location;
LocEqual:	location × location	→ boolean;
LocX:	location	→ real;
LocY:	location	→ real.

Sample definitions

LocEqual(LocCreate(x_1, y_1), LocCreate(x_2, y_2))
→ if (($x_1 = x_2$) and ($y_1 = y_2$)) then
True
else
False.

LocX(LocCreate(x, y)) → x.

Data type Image

Operations

∘ImageCreate:	imagetype × boolean × location × {local parameters}	→ image
ImageLocate:	image × imagepart	→ location

The *Create* image operation adds a new image instance to the set of active images, I. Because of the different types of images, the specific local parameters to the function may vary. For example, creation of a circle image may take real values for the radius and fill style and a color parameter for the image color. The *Create* operation also adds the corresponding new element, *im*, to the configuration C as follows:

$$(Create \rightarrow im) \otimes C \;\Rightarrow\; C = \text{cons}(im, C).$$

This simply states that the image is added to the front of the configuration list.

Note that the *Create* operation does not append the resulting configuration to the algorithm animation list. That is, images added with the *Create* image operation do not appear in the animation viewing window until a subsequent animation frame is generated by a *Perform* transition operation.

generators, and are a sufficient set of operations within a type to generate any object of the type.

Operations are denoted and defined in the following manner: Basic generator operations are identified by the "$\circ$" symbol preceding the operation name. Generator operations are defined by expressions consisting only of basic generators and simple algebraic manipulations, along with basic constructs such as *if-then-else* and *while-do*. For manipulations of objects such as paths and transitions consisting primarily of lists, we also utilize the list operations *car*, *cdr*, *cons*, *append*, and *reverse*. Inquiry operations are defined by providing their result upon objects generated by all of the basic generators of that data type.

Both generators and basic generators affect the animation state by adding a new object to a field in the animation state as discussed earlier. These manipulations can be defined as in the location example

$$Create(x, y) \rightarrow lo[x, y] : (lo \in L) \cup S,$$

where L is the location component of the animation state S. The equation merely defines that the *Create* location operation adds a new location object with the given x and y values to the location component of the animation state. All generator operations within the paradigm function similarly, so we will omit these manipulations from the operation definitions. Note that inquiry operations do not modify the animation state at all.

We believe that the descriptive methods we have chosen are rigorous enough to fully detail the data type operations, and are informal enough so as not to obscure what is really going on. The conditional

if (x)

where x is a list of a given type of object, returns true except when x is the null list. The symbols $\langle\ \rangle$ denote a list of some object, and the symbol $\emptyset$ denotes an empty list. When equality between real numbers is checked, a small built-in tolerance level is used. For reasons of brevity, we provide a flavor of the definitions for the location, image, and path data types below, and we focus on the transition type because it is the most interesting and important. For complete definitions, see [9].

the same image, one red and one blue, exist in a moment, their application ordering is critical. Whichever one is applied last will determine the image's color in the subsequent configuration.

Fortunately, because the only method for combining transition units into a moment is the *Compose* operation, this issue is resolved by defining the way *Compose* generates transition unit lists. We use the simple protocol that when two transitions are composed, the transition units in the first transition (parameter) precede those of the second when forming moments. That is, given the operation $TransCompose(tr_1, tr_2)$ where tr_1 changes an image to red and tr_2 changes an image to blue, following the performance of this composition transition, the image will be blue.

The second important semantic issue concerns the ordering of images in a configuration and subsequently in the viewing window. As noted previously, images at the beginning of the configuration list will appear to be closer to the viewer. But how do images establish positions within the configuration list? The problem is resolved through two simple rules.

1. Whenever an image is created, it is added to the front of the configuration list.

2. Whenever a transition unit of type *raise* or *lower* is applied to an image, the image is moved to the front or tail of the configuration list, respectively.

Hence, barring any use of the *raise* and *lower* transitions, the initial creation order of images defines their relative ordering in viewing planes of the animation coordinate system.

Below we provide an introduction to the semantic formalisms used to specify the data types and operations of the path-transition paradigm. The abstract data types are specified as follows:

Location	*lo*	x:Real, y:Real
Image	*im*	*type*:ImageType, *pos*:Location, *vis*:Boolean, {*local parameters*}
Path	*pa*	*offsets*:list(Location)
TransUnit	*tu*	*type*:TransType, *image*:Image, x:Real, y:Real
Transition	*tr*	*moment*:list(list(TransUnit))

In order to specify the data type operations, we utilize techniques similar to those of Mallgren[7]. Following his axiomatic designations for describing operations upon graphic data types, we divide operations upon the animation data types into three categories. *Generators* produce objects of the type of the operation (location objects for location operations, etc.). *Inquiry* operations return objects of some type other than that of the operation. Typically, they examine an object of the type of interest and return some parameter about it. Finally, *basic generators* are subsets of the

animation coordinate system, i_1 will obscure i_2 if i_1 it is earlier in the configuration list.

Transitions that are performed in the animation paradigm provide the intermediate points in time for identifying a particular configuration as a frame in the animation. The lists of transition units that occur "simultaneously" between frames generate a new configuration from the current one. (Recall that we defined a moment as a list of transition units corresponding to the changes that occur between two frames of the animation.) Within the paradigm we define the formal notion of an *animation* as an ordered list of configurations. Therefore, we capture the essence of an entire animation by this simple concept, a list of image lists.

The sole semantic purpose of each of the data type operations except for the image *Create* and *Perform* transition operations is to modify the animation state. Most operations add a new location, image, path, or transition instance to the corresponding field in the animation state. Some simply return a parameter value about a data type instance and make no change to the animation state. The *Create* image function, in addition to modifying the animation state, adds images to the front of the image configuration list. The *Perform* transition operation is even more complex, however, as it can modify the animation state, reorder the image configuration list, and add one or more of these new configurations to the animation list. The number of new configurations generated corresponds directly to the length of the path(s) that serves as an argument to the transition being performed.

Two related, primary semantic issues that arise in the paradigm are the ordering of transition units within a moment of a transition, and the ordering of images within the configuration. These issues are important because we want to provide consistency throughout the paradigm and its operations.

Because of the *Compose* operation, many transition unit actions can occur "simultaneously" in one moment of a transition between the frames of an animation. The appearance of simultaneity is given by displaying the images in the current configuration, modifying the images' parameters one-at-a-time as dictated by the transition units in that moment, then redisplaying the images in the resulting configuration. The difficulty occurs in that at the lowest level the transition units must still be applied in a serial manner within the moment. The serial application of two *move* transition units defined on an image present no problem. Either application ordering of the two results in the image ending up at the same position. But if two *color* transition units defined on

by the particular transition unit lists in a moment all occur "simultaneously" between the frames of the animation. By processing moments, one after another, the frames of the animation are generated. Specifics of the semantics of *Perform* are presented in Section 4.

3.4 Animation Scenes

For manageability reasons, we decompose an animation in the path-transition paradigm into a set of smaller logical components called *animation scenes*. The creation and breakdown of an animation into scenes is purely determined by the animation's designer; there is no "correct" decomposition. Quite often, however, a logical separation of important actions presents itself, not unlike the breakdown of a large computer program into subroutines.

We model an animation scene informally as a parameterized procedure. The parameters to scenes are received by value, and they consist of integer, real or string values. The procedure body consists primarily of the data type operations described earlier. The use of mathematical and control flow facilities is also often useful. Because we provide an animation paradigm rather than a stand-alone formal language implementation, embedding the data type operations in an imperative language such as Pascal or C is practical and sufficient.

4 Semantics of Animation Operations

We model an animation in the path-transition paradigm through an *animation state* defined by the 4-tuple (L, I, P, T), where L is a set of locations, I is a set of images, P is a set of paths, and T is a set of transitions. These four sets represent the active available objects during an animation.

Note that at any particular point in time the set of active images, I, does not uniquely determine the appearance of images residing in the animation coordinate system. For example, the set of images could include two rectangles, one green and one yellow, each with the same position and size. Depending upon the relative ordering or layering of the rectangles, we could see green or yellow at that position. Consequently, we utilize the notion of "display priority" to help define the *configuration* of images, C, which is an ordered list of all the active images in the set I. Given two images, i_1 and i_2, if i_1 precedes i_2 in the configuration list, then i_1 is considered to be on a viewing plane closer to the animation viewer than i_2. That is, if i_1 and i_2 occupy the same position in the

⟨ ⟨ (*move*, i_1, 0.2, 0.4) ⟩, ⟨ (*move*, i_1, -0.1, -0.3) ⟩, ⟨ (*move*, i_1, -0.4, 0.1) ⟩ ⟩	moments	⟨ ⟨ (*resize*, i_1, 0.1, 0.0), (*fill*, i_2, 0.1, 0.1) ⟩, ⟨ (*delay*, i_3, 0.0, 0.0), (*color*, i_3, 0.2, -0.04) ⟩, ⟨ (*move*, i_3, -0.5, 0.0) ⟩, ⟨ (*visibility*, i_2, 0.0, 0.0) ⟩ ⟩

Figure 4: The left transition is simple; the general transition on the right has been built up using the *Compose* operation. The notation ⟨ ⟩ denotes a list.

number of moments are composed, the shorter transition is implicitly padded at the end with empty moments to make the composition one-to-one. A good way to intuitively think about the transition operations is that concatenation combines lists of transition moments end-to-end or bottom-to-top, whereas composition combines them side-to-side.

As a more sophisticated example of composition, suppose we wish to define a transition that moves an image i along an existing path p_1 of length 20, with the image's visibility toggling during the middle 10 movements in the path. Let us define the following notation for the transition operators:

$t_1 \cdot t_2 \equiv$ concatenate transitions t_1 and t_2 (t_1 occurs first).
$t^{num} \equiv$ iterate the transition t num times.
$t_1 \odot t_2 \equiv$ compose transitions t_1 and t_2.

By creating the paths

$$p_2 = \mathit{Null}(5),$$
$$p_3 = \mathit{Null}(1),$$

and the transitions

$$t_1 = \mathit{Create}(\mathit{delay}, i, p_2),$$
$$t_2 = \mathit{Create}(\mathit{visibility}, i, p_3),$$
$$t_3 = \mathit{Create}(\mathit{move}, i, p_1),$$

the desired animation is simply the following expression:

$$(t_1 \cdot t_2^{10} \cdot t_1) \odot t_3.$$

Once a transition has been defined, the *Perform* operation carries it out. Intuitively, this command works by sequentially examining the moments making up a transition; the changes denoted

transition type, i is the image affected, and x and y are modifier values taken from a path offset. Simple transitions are just ordered lists of transition units.

Operations upon transitions are listed below:

Create:	Create a new transition instance (type, image, path)
Concatenate:	Create a transition consisting of one action occurring after another
Iterate:	Create a transition that is a repetition of an action
Compose:	Create a transition that consists of two actions occurring simultaneously
Perform:	Make an action happen now

The *Create* operation defines simple transitions for subsequent use. The *Concatenate*, *Iterate*, and *Compose* operations combine transitions to form new, more complex transitions. The *Concatenate* operation binds two transitions together into a single, new transition which corresponds to the latter transition argument commencing immediately after the initial transition argument is complete. That is, the two lists of transition units are appended to form one longer list. The *Iterate* operation performs repeated concatenation of a transition with itself. The third transition operation, *Compose*, is more complex.

Composition introduces a form of concurrent execution of transitions. When two transitions are composed, the animation actions they denote occur simultaneously. Consider two separate *move* transitions defined upon a circle and a line respectively, that each utilize the same path argument. Composing these transitions creates a new transition that exhibits each of the images moving by the same relative offset, as defined by the path, between each frame of the animation. In essence, between each animation frame two transition units will occur—one for the circle image and one for the line image.

Let us define a transition *moment* as a list of transition units that occur "simultaneously" in one time unit, in order to generate a new frame of an animation. This permits a formal definition of a general transition as an ordered list of moments. Notice that a simple transition contains moments with only one transition unit. Figure 4 provides examples of these different types of transitions.

The concurrency exhibited by *Compose* is accomplished by combining corresponding moments in the two transition arguments into a new, larger moment in the resulting transition. The two lists of transition units making up the first moment of each transition being composed are appended to form a longer transition unit list that is the first moment of the resulting composition transition. The process is repeated for all moments in the transitions. If two transitions containing an unequal

suggested by their names, and the *Compose* operation combines paths on an offset by offset basis, much like vector addition. Note that our use of the word "compose" refers to a specific type of combination, not the formal mathematical definition of composition.

3.3 Transition

A transition provides an animation with action by utilizing a path parameter to modify the attributes of an image, such as its position or appearance. Moving a line in a wavy path, altering the fill style of a circle, or shrinking a rectangle as it changes color are all examples of transitions. Additionally, we can think of the simultaneous occurrence of all three of these actions as a new, albeit more complex, transition.

A *simple* transition is defined by the 3-tuple (t, i, p), where t is the transition type, i is the image being altered, and p is a path argument modifier. Some common examples of simple transition types are *move*, *resize*, *fill*, *color*, *raise*, *lower* and *alter visibility*. Like that done with primary images, simple transitions are not restricted to a specific set of transition types. The general model of a simple transition then allows the paradigm's implementation to be extended when a new type of transition is needed.

The path data type plays an important role in the definition of a transition. As discussed earlier, the coordinate pairs making up a path correspond to the individual frames of an animation. In essence, the path argument defines the duration of a transition. Nevertheless, different types of transitions utilize path arguments in different ways. *Move* transitions use paths as routes for images to follow. Each coordinate pair of the path specifies the relative offset that the image should move for the next animation frame. *Visibility* transitions disregard the specific x and y values of the coordinate pairs of a path; each coordinate pair is interpreted as toggling the image's visibility in the next animation frame. *Fill* transitions utilize the x component of a coordinate pair by adding its value to an image's fill value, which ranges between 0.0 (outline) and 1.0 (completely filled).

By decomposing a path into its offsets and considering each along with the transition type and primary image components, the substructure that makes up a simple transition becomes evident. This structure primitive, called a *transition unit*, is an atomic level action that occurs in an animation. Each transition unit defines a particular action to occur prior to the next frame of the animation. More formally, a transition unit is defined as a quadruple (t, i, x, y), where t is the

Figure 3: The movements of the three basic path types. Each is defined to have length 20.

motions in simple animations designed to run in square animation windows with a side measuring 1.0 units. The *MakeType* operation creates a path in one of these types. Although the paths created by this operation can be used directly, most often they will be edited in some way by a path modification operation before actually being used.

The *Interpolate* operation adds or removes offsets from a path while maintaining its overall change in x and y. This operation can be used to control the relative speed of an action. Changes occur more quickly along a path with fewer offsets. The *Color* operation returns a path of length one, which when interpreted in a *color* animation action, will change an image to a specified color.

One key issue in using paths in an animation occurs when using paths as routes for image movement. This type of action often takes the form of moving an image from its current location to some new location that depends upon particular parametric data. These two locations, the starting and ending points, typically cannot be determined until the particular instant before the movement action occurs. Consequently, precompiled paths that were defined at animation design time, such as those from the *MakeType* operation, may not move the image to the desired position. Therefore, the paradigm includes operations that receive two locations and create a path that begins at the one location and ends at the other.

Distance is such an operation. It receives two locations and a distance parameter, from which it creates a path that moves in a straight line from source to destination and that moves the given distance between offsets. The *Example* operation also uses source and destination locations like *Distance*, but its parameter for creating a new path is an example path. The new path is designed to be similar to the example path; this involves maintaining the same path length and the same geometric characteristics in moving from the source to the destination location.

The *Concatenate* and *Iterate* path operations append copies of paths together in the manner

control the relative spacing between offsets in a path in order to control the smoothness of an animation. In a graphical environment that supports fast image updates, utilizing paths with very small changes in x and y from offset to offset produces smooth, visually pleasing animation effects. Removing intermediate offsets speeds up a motion; adding offsets slows it down. Operations upon paths follow below:

Create:	Create a new path given a list of offsets
Length:	Return the length of a path
Dx:	Return the change in x a path makes
Dy:	Return the change in y a path makes
Null:	Return a path with a given number of (0.0,0.0) offsets
MakeType:	Return a path in one of three predefined types
Copy:	Return a copy of a given path
Rotate:	Return a path that is the rotation of another
AddHead:	Return a copy of another path with null offsets at the head
AddTail:	Return a copy of another path with null offsets at the tail
DeleteHead:	Return a copy of another path with offsets removed from the head
DeleteTail:	Return a copy of another path with offsets removed from the tail
Scale:	Scale a path by values in x and y
Extend:	Add given x and y values to each offset in the path
Interpolate:	Alter the number of offsets in a path, but maintain the motion
Color:	Make a path that will change an image to a given color
Distance:	Make a straight path with offsets moving a given distance
Example:	Make a path, similar to another, that moves between two given locations
Concatenate:	Append paths to make a new path
Iterate:	Repeatedly append a path to itself
Compose:	Combine paths on an offset by offset basis

The basic operation for developing paths, *Create*, receives an ordered list of locations; each location defines the x and y values of subsequent offsets. Although the *Create* operation can produce all possible paths, it is an inconvenient format for defining paths that will actually be used in the animation sequences. For additional utility, we provide operations that create some basic types of paths by introducing other, more pragmatic, path creation operations. The *Null* operation returns a path with the given number of null (0.0,0.0) offsets. It is useful for defining paths with a desired length in which individual offset values do not matter.

Because most image movements follow routes such as straight lines or simple arcs, the paradigm includes three predefined basic path types: *straight*, *clockwise*, and *counterclockwise*. Visualizations of the three path types are presented in Figure 3. A standard for these basic types has been defined so that each has length 20 and moves 0.2 distance units in x from the starting location to the ending location of the path. These values are somewhat arbitrary—they produce aesthetically pleasing

Another issue involving images is their relative layering in the animation coordinate system. If two rectangles of differing colors occupy the same location, we must determine which one will be considered to be on a plane closer to the viewer, and hence be visible. To establish consistency in the paradigm, we set up a relative ordering of images in order to handle this problem. When a new image is added, it is always placed in the top-most relative plane (closest to the viewer). We also include animation actions in the paradigm that place images in the top-most and bottom-most planes. We will further discuss these types of consistency issues in the next section concerning paradigm semantics.

3.2.1 Path

In the path-transition paradigm, a path is the only object used in an action in order to modify the attributes of an image from frame to frame. Using a path as a directional route for image movement, modifying the image's position, is a natural notion. An important concept within our paradigm, however, following the style of Baecker's path-descriptions, is that a path also can be interpreted to modify values of color, size, fill, and so on. By establishing a consistent model for an animation action, each modifies an image using a path parameter, but nothing else, the design of animation actions is simplified. Animation designers need not worry about remembering the exact list of parameters that a particular action requires.

Formally, a path is defined to be a finite ordered sequence of real-valued (x, y) coordinate pairs, where each pair designates a relative offset from the previous position. The initial coordinate pair of a path is offset from the path's starting position. The *length* of a path is defined as the number of coordinate pairs it includes.

Paths designate two-dimensional routes in an abstract coordinate system. For actions such as movement, we can think of the coordinate system coinciding with the animation coordinate system. It is important to understand, however, that the offsets in a path are not absolute coordinate locations. Once a path is created, it is used in an action to apply to an image that has an absolute position. Particular relative offsets of the path then define how the image will change in the animation relative to the image's absolute position.

A path also contains a relative time component because each path offset determines the location and appearance of an image in a subsequent frame of the animation. Animation designers can

```
rectangle 0.0 0.0 0.3 0.6 black 0.0
circle 0.15 0.1 0.05 red 0.2
circle 0.15 0.3 0.05 yellow 0.2
circle 0.15 0.5 0.05 green 1.0
```

Figure 2: A composite image definition representing a traffic light that signals "go."

one another. The ability to coalesce a set of primary images into one new image type and repeatedly create instances of the image extends the graphical display of the paradigm and provides a practical mechanism for defining complex imagery. Composite images are defined as a list of primary images in a local coordinate system. Positions of the primary images are all relative to the position parameter provided as the location of the composite image. For example, suppose we wish to repeatedly create an instance of a traffic light in an animation. The definition in Figure 2 produces the accompanying composite image. Each line of the definition identifies a new primary image. The first two real numbers on a line indicate the location of the image; the next numbers indicate width and height for the rectangle or the radius for the circles. The final number represents a fill percentage, with 0.0 being outline and 1.0 being solid color. Operations upon images follow:

Create:	Create a new image instance
Locate:	Return the location of a part or extremity of an image

Both primary and composite images originate via the operation, *Create*. The image data type also includes a bounding box oriented image part query operation, *Locate*. Given an image and a compass position such as *NorthWest* or *East*, or the special *Center* position, the operation returns a location object that corresponds to that current position of the image. This operation is useful as illustrated by the following example. Suppose two rectangle images exist, and we want to create an animation in which the first rectangle moves over on top of the second rectangle, like boxes stacked up, with their left edges aligned. To do this, we would identify the *SouthWest* part of the first rectangle and the *NorthWest* part of the second rectangle. Then, to carry out the desired animation, we would create a movement path for the first rectangle to traverse with endpoints at these two locations.

two styles of graphical objects called *images*, exist: *primary images* and *composite images*.

Primary Image

A primary image is the most basic style of image manipulated in the paradigm. Some common examples are lines, circles, rectangles, polylines, polygons, ellipses, splines and text. Requiring that primary images conform to a specific format or including a predefined set of primary image types would restrict the animation paradigm. Consequently, we provide a general model of a primary image.

Formally, we define a primary image by a set of parameters and a set of methods. A primary image as a 6-tuple (t, o, v, P, R, d) where t is the primary image type, o is the image's position, v is the image's visibility, P is a set of parameters local to the image type, R is a set of transition methods, and d is a drawing method. The use of the local parameters P is necessary because specific types of primary images differ. For example, a line may have *size*, *thickness*, and *color* local parameters, whereas a circle may have *radius*, *fill*, and *color* parameters.

Note that some overlap exists in our example primary image types. For instance, polygons include rectangles, polylines include lines, and so on. We provide the more specific image types such as lines and rectangles for practical reasons. The workstation graphics packages that will be used to implement the paradigm often include optimized routines for manipulating these specific image types. The simpler images such as lines and rectangles are also the most commonly used images in animations, thereby magnifying their importance.

Images are modified by transition data type instances (explained in a following subsection) acting upon them. As part of a primary image definition, each image type must have a method that defines how the different transitions affect the image's parameters. Movement and visibility transitions affect each primary image type in the same manner: the position and visibility fields are modified. Other transition types may affect images differently. For example, a circle may have its radius modified by a *resize* transition, but a rectangle may have its width and height changed. The final component of the primary image definition is a draw method that translates a primary image from its parametric definition to a static graphical object by using the drawing routine of the paradigm's particular system implementation.

Composite Image

A composite image is a collection of primary images usually having some geometric relationship to

Figure 1: Visual depictions of the four abstract data types in our animation paradigm.

of animations we desire.

3.1 Location

A *location* is a position of interest within the animation coordinate system and is identified by an (x, y) coordinate pair. The ability to save and reference particular locations is valuable because geometric layout is an important component of animation design. The operations defined for the location data type are listed below:

Create:	Allocate a new location given x and y coordinates
Modify:	Change the values of a particular location's coordinates
Equal:	Check if two locations reference the same coordinate (uses small tolerance)
X:	Return the x coordinate of a particular location
Y:	Return the y coordinate of a particular location

3.2 Image

Computer animation is the process of simulating continuous movement and action by repeatedly making changes in the location, size, color, etc., of graphical objects. In our animation paradigm,

transformations, which allowed many of the operations and manipulations common to graphics programs to be specified formally using only basic operations and algebraic axioms[7]. By embedding the graphic data types in the language Pascal, he also created a form of graphics language that was powerful enough to build arbitrary hierarchical line drawings. We have adopted some of the techniques Mallgren introduced, and we have extended them to encompass animation.

Fiume utilized a somewhat different approach to formal specifications for a non-trivial graphical object by attempting to specify formally a *bitmap* object and its relationship to an *image* object[8]. In his approach, these objects were modeled as abstract data types with each having lists of operations to be performed on them. Citing the difficulty of an informal language description, Fiume used the formalisms of mathematics and sets to specify the operations, a method somewhat more complex than the algebraic methods used by Mallgren.

3 The Path-Transition Paradigm

A computer animation consists of a collection of graphical objects to which a collection of modifications such as movement, changes in color or size, and so forth, occur over time. We have identified a set of four abstract data types that encapsulate the constituents of an animation. These abstract data types serve as the basis for our animation paradigm. They include the graphical *images* on the screen, the *locations* they occupy, the *transitions* the images make, and the *paths* that modify the image transitions. We call this method the *path-transition paradigm*, based on the two data types that drive the animation actions. As part of each data type, we also provide a set of constituent operations. Defining an animation involves creating and manipulating specific instances of the data types using these operations. Figure 1 provides a visual depiction of the data types.

Images in our paradigm reside in a real-valued, infinite *animation coordinate system* that allows animations to function regardless of the animation window size. In practice, our paradigm is designed to assist the production of animated views in windows on a bit-mapped graphics workstation. Therefore, the particular system implementation of the paradigm must map the animation coordinate system onto an actual workstation display window.

In the remainder of this section we will discuss each of the four abstract data types in more detail. We also will list the operations that we have identified as being useful for creating the styles

responded to a tick) and the vertical dimension specifying individual animation actions.

The ASAS animation control language of Reynolds was a LISP-based design mechanism for specifying computer graphics animations using an *actor-script* metaphor[2]. Actors usually consisted of a visible object (polygonal data definition) and an associated "chunk" of animation code that was executed once for each frame, and they coordinated actions with other actors through message passing thereby coordinating their actions.

Fiume, Tsichritzis, and Dami presented an object-oriented, temporal scripting language for expressing the coordination of animated objects[3]. Their approach involved the use of a library of animation objects and a library of motions available to the objects. The scripting language facilitated control of object behaviors via facilities for motions' duration, activation, and termination of types sequenced, simultaneous, and delayed, as well as constructs for repetition, asynchronous execution, temporal scaling, non-determinism, and temporal overlap.

Arya employed a markedly different approach to designing animations in which simple keyframe animations were designed using the functional programming language ML[4]. His system included a *picture* atomic data type and a compact set of primitive operations. By using higher-order operations, pictures could be combined to form *movies*, which also included higher-order spatial and temporal combination operators.

In our paradigm for designing animations, objects follow two-dimensional tracks or paths in the animation coordinate system. This methodology of creating animation sequences has been termed "path-of-motion[5]." The use of paths to control an animation is a natural and tested notion. The original development of this methodology involved the *path-description* and *p-curve* concepts of Baecker[6]. A path-description consisted of a sequence of values in an arbitrary coordinate space that was used to modify the attributes of an animation. A p-curve, the visible manifestation of a path-description, was a continuous trail of symbols placed along a two-dimensional route, corresponding to the sequence of values in a path-description.

Another topic relevant to our work is graphical specification. Although the merits of specification are universally regarded as important, relatively little work has surfaced concerning specification of picture images, especially animated images.

Mallgren provided one of the earliest efforts toward formal specification of graphical objects by introducing special *graphic data types* such as points, regions, geometric functions, and graphic

The use of animation is now becoming evident in a variety of areas such as computer-aided design, user interfaces, and on-line help information. The initial application area of our work is *algorithm animation*, the production of dynamic visualizations of the abstractions, data, and operations in executing computer programs. In an algorithm animation, an executing program provides information about its computation state to the accompanying dynamic view(s), thereby controlling the imagery and display. The animation frames are generated sequentially in real-time, without excessive delays from image rendering. Because the run-time data from a program dictate how the animation should appear, an algorithm animation cannot be totally predefined. Essentially, the designer implementing the animation provides an outline for the action which is then completed by the program's run-time data, operations, and behavior.

The next section of the paper describes related work. Section 3 introduces the path-transition paradigm and its four abstract data types. Section 4 contains semantic specifications of the data types and their operations. Section 5 describes our particular application area, algorithm animation. Section 6 illustrates an example animation, the Towers of Hanoi problem, and how it is built using the paradigm. Finally, Section 7 discusses strengths and limitations of the work, and lists future directions of research.

2 Related Work

The ideas discussed in this paper relate to a variety of topics such as animation languages, animation methodologies, and graphical specification. Many animation languages have been developed for controlling sophisticated three-dimensional animation sequences. These languages are impractical for the objectives described earlier because, for the most part, they have a steep learning curve and they utilize a considerable amount of time to generate animation frames. Typically, their extensive rendering requirements force animation frames to be stored and replayed in the desired animation sequence only after all have been generated.

The Dial animation language by Feiner, Salesin, and Banchoff utilized a simple, two–dimensional notation to let animators express parallelism[1]. Time was described as a sequence of small, finite-length ticks, and animators expressed the coordination of an object's motion using a two-dimensional, chart-like syntax with the horizontal dimension representing time (each column cor-

1 Introduction

The increasing availability and sophistication of workstations has made programmers come to expect more advanced imagery in user interfaces and programming environments. With improvements in display hardware, animation is now a viable part of those interfaces. Unfortunately, the software developer searching for a practical, two-dimensional, color animation toolkit for his or her workstation does not have much to choose from. Current animation design methods are usually ad-hoc, lacking formality, and peculiar to a particular application. Often, such animations only consist of a repeated shuffle through slightly altered sequences of precomputed bitmaps.

To meet these needs, an animation toolkit with the following capabilities is desirable:

- **Data-driven:** The animation should be encapsulated in procedures that can receive parameters which control the animation's format. For instance, parameters could supply the object to be moved, the destination location of an object's movement, the size that an object should grow to, etc.

- **Real-time performance:** Animation generation should occur "on-the-fly." That is, the generation of animation frames should be fast enough so that a visually pleasing animation occurs in real clock time. Excessive delays resulting from rendering are not desirable. These performance capabilities also should be achieved in a widely available environment such as a workstation running UNIX and the X11 Window System.

- **Continuous imagery updates:** The animation package should contain a methodology that facilitates designers implementing smooth, aesthetically pleasing transitions between animation states rather than jerky, snap-shot view updates.

- **Easy-to-learn:** The complexity of designing and implementing animations should be kept to a minimum. Succinctly stated, the simpler it is to produce an animation, the better.

In this paper we introduce the *path-transition paradigm* for creating color, two-dimensional, real-time animation sequences. The paradigm supports aesthetically pleasing, smooth image changes, and includes animation primitives specifically tailored to simplify animation design and implementation. Our approach is based upon four abstract data types: location, image, path, and transition. By creating and manipulating instances of the data types, designers produce their intended animation actions. Consequently, the paradigm exhibits an object-based appearance. We also develop formal models and precise semantics of the data types and their operations in order to insure consistency and formality.

Abstract

Animation is becoming an increasingly popular feature of user interfaces and software development environments. Unfortunately, few tools for adding animation to interfaces and environments currently exist, and those that are available tend to be ad-hoc approaches, lacking any formal semantic basis. In many of these cases, animation is achieved by the simple shuffling of bitmap images.

In this paper we describe a practical animation paradigm for creating real-time, two-dimensional, aesthetically-pleasing, color animations. Our method is called the *path-transition paradigm*, and it is based on four abstract data types: location, image, path, and transition. Producing animation sequences in this paradigm involves the creation and modification of instances of the data types. We develop precise semantics and specifications for the data types, resulting in a rigorous framework for describing the actions that occur in an animation. We present results from our experiments with the paradigm applied to an algorithm animation system that displays sophisticated views of program executions.

The Path-Transition Paradigm: A Practical Methodology for Adding Animation to Program Interfaces

John T. Stasko

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

(404) 853-9386

email: stasko@cc.gatech.edu

June 10, 1991